Compact Representation and Efficient Manipulation of Sparse Multidimensional Arrays

Timoteus Dahlberg
Abstract

Efficient manipulation of sparse multidimensional arrays, or tensors, is of interest because their decompositions have applications in many different areas. These areas include neuroscience, machine learning, psychometrics, data mining, numerical analysis, and more. This thesis aims to develop the performance-critical parts of a library for manipulating sparse multidimensional arrays by focusing on sorting them in one or more dimensions—a fundamental operation on which many other operations can be built.

High performance is achieved by tailoring algorithms to a compact representation scheme. Evaluation is done on different algorithms and implementation techniques. The result is shown to be 20 to 70 times faster than qsort in the C standard library. The resulting library is open source.
1 Introduction

A multidimensional array is a mathematical object which can be manipulated using different algorithms. These algorithms often require efficient access to elements in some particular order. A multidimensional array can be compactly represented in memory using special formats. The way that they are stored affects the performance of algorithms manipulating them. One of the goals of this thesis, presented in Section 1.2, is to access these elements efficiently using a special storage format. Section 1.1 explains how this is achieved before the purpose is presented in Section 1.2.

1.1 Background

A multidimensional array $A$ with $N$ dimensions is an object whose elements are described by a multi-index $(i_1, \ldots, i_N)$ consisting of $N$ indices $i_n$. The two-dimensional array $A$ in Equation 1.1 is sparse and consists of numerical values. A sparse multidimensional array has most of its elements equal to zero. A suitable format can be used to represent them in a compressed, memory-efficient, way. Bader and Kolda [1, ch. 3.1.1] argues that the coordinate format is well suited for representing sparse multidimensional arrays due to its simplicity and flexibility compared to other formats. Sparse multidimensional arrays can, using the coordinate format, be represented as two one-dimensional arrays of the same length: one array, $\text{val}$, for storing all the non-zero elements and another array, $\text{omega}$, for storing the corresponding multi-indices. This can be seen in Equation 1.1 where each row of the $\text{omega}$ array represents one multi-index.

$$
A = \begin{bmatrix}
0 & 4 & 0 \\
0 & 2 & 0 \\
0 & 0 & 7
\end{bmatrix} \quad \text{val} = \begin{bmatrix}
4 \\
2 \\
7
\end{bmatrix} \quad \text{omega} = \begin{bmatrix}
(0, 1) \\
(1, 1) \\
(2, 2)
\end{bmatrix}
$$

The term tensor is synonymous with multidimensional array and these terms will be used interchangeably throughout this thesis. The dimension, $N$, of a tensor is called order and higher-order tensors are tensors where $N \geq 3$. Decompositions of higher-order tensors have applications in neuroscience, machine learning, psychometrics, data mining, numerical analysis, computer vision, and more [2, 3]. Algorithms that operate on sparse tensors frequently require efficient access to elements in some particular order [4]. This can be achieved by sorting the array $\text{omega}$ and correspondingly permute the array $\text{val}$.

The performance of these algorithms on today’s hardware is often limited by memory accesses as opposed to CPU cycles, increased memory efficiency at the cost of CPU cycles can therefore lead to increased performance. Each multi-index can be compactly represented by packing the binary representation of each index using bit fields. Suppose that a tensor $A$

---

1 Zero-based indexing is used, so $i_n$ is in the range $0 \leq i_n < I_n$.
2 Formats such as compressed sparse row (CSR) or compressed sparse column (CSC).
is of order 3 and size $3 \times 10 \times 6$. The first index can be represented using $\lceil \log_2 3 \rceil = 2$ bits, the second index by 4 bits, and the third by 3 bits. Let $a_1a_0$ denote the binary representation of $i_1$, $b_3b_2b_1b_0$ the binary representation of $i_2$, and $c_2c_1c_0$ the binary representation of $i_3$. The binary representations of the indices can then be packed into a 16-bit word as seen in Table 1.1. The specific multi-index $(2, 7, 3)$ for the same tensor $A$ can also be seen in Table 1.1.

<table>
<thead>
<tr>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>bit number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>$(2, 7, 3)$</td>
</tr>
</tbody>
</table>

Table 1.1: The binary representation of the multi-index $(2, 7, 3)$ for a packed tensor $A$ of size $3 \times 10 \times 6$.

The multi-indices of $\omega$ can consequently be packed into bit fields using $b$ number of bits. The reasonable assumption is made that this packing can be done using $b \leq 32$ bits, even though it restricts the maximum size of the tensor. This means each multi-index fits in a 32-bit word. It follows from this assumption that the number of multi-indices and their values in the $\omega$ array will be between 0 and $2^{32} - 1$.

Frequently, one not only desires the sorted array $\omega$ as output but also the actual permutation that was applied by the sorting operation. This is useful, for instance, to avoid permuting the $val$ array more than once, or to construct multiple simultaneous views into the same $val$ array, each sorted by different criteria. To obtain the permutation, each multi-index can be augmented by another 32-bit word as the high-order bits of a 64-bit word. Before sorting the multi-index array $\omega$, the 32 high-order bits of $\omega[k]$ can be set to the binary representation of $k$. After sorting, the high-order bits of $\omega[k]$ will denote the original position of the corresponding element in the $val$ array and in that sense represent the mapping from an output position back to the corresponding input position. Since 64 bits matches the native word-size of 64-bit processors, packing them as such leads to increased memory efficiency even though manipulating them may require additional instructions.

Equation 1.2 shows a multi-index set, $\omega$, sorted according to different dimensions of tensor $A$ in ascending order. Each multi-index consists of a set of keys, each key corresponds to an index in the tensor. The multi-index set is sorted according to specific dimensions by sorting each corresponding key of each multi-index. In Equation 1.2, each multi-index consists of three keys and each key is represented by an integer. The first multi-index set, $\omega_{012}$, is sorted according to dimensions $0, 1, 2$ while the second, $\omega_{21}$, is sorted according to dimensions $2, 1$.

$$\omega_{012} = \begin{bmatrix} (0, 2, 5) \\ (0, 9, 3) \\ (0, 9, 5) \\ (1, 7, 3) \\ (2, 1, 0) \\ (2, 9, 5) \end{bmatrix} \quad \omega_{21} = \begin{bmatrix} (2, 1, 0) \\ (1, 7, 3) \\ (0, 9, 3) \\ (0, 2, 5) \\ (2, 9, 5) \\ (0, 9, 5) \end{bmatrix}$$  \hspace{1cm} (1.2)

Since each multi-index is packed, as shown in Table 1.1, each key must be extracted from the bit fields prior to being sorted. This is what distinguishes this problem from generic sorting problems. After the relevant keys have been extracted, the multi-index sets can be sorted in the same way as shown in Equation 1.2. There are different ways of extracting keys.
from the packed multi-index sets, some of which are discussed in Chapter 2 and evaluated in
Chapter 3.

1.2 Purpose and aims

This thesis aims to identify fast sorting algorithms on today’s computers and customize them
for 32-bit multi-indices accompanied by 32-bit permutations having a special non-standard
format as described in Section 1.1. The long-term goal is to develop a high-performance
library for manipulating sparse multidimensional arrays. A fundamental operation in such a
library, on which many other operations can be built, is sorting in one or more dimensions.
The purpose of this thesis is to develop the foundation of this library with focus on sorting.

1.3 Outline

The rest of this chapter summarizes a literature survey done to identify efficient sorting
algorithms that can be adapted to the particular sorting problem at hand. The goal of
the survey is to identify algorithmic ideas and implementation techniques that have been
demonstrated to work well in practice. In Chapter 2 some of the more promising ideas are
used to synthesize novel solutions to the special sorting problem. The implementations are
then evaluated in Chapter 3.

1.4 Literature survey

Sorting algorithms are often classified as either comparison-based or noncomparison-based.
The former are general purpose and sort elements using pairwise comparisons. The latter
sort elements by partitioning them based on their values. It is well known that the latter
has better asymptotic time complexity than the former, which has a time complexity of
Ω(n \log_2 n) [5, ch. 5.3.1]. This does, however, not say which class of algorithms are superior
for a certain problem of finite size, since the time complexity is asymptotic.

Sorting algorithms can be in-place which means they only require a constant amount of
extra memory. Sorting algorithms can also be stable which means they preserve the relative
order of elements with equal keys. Algorithms that are not stable are called unstable.

Radix sort [5, 6] is a noncomparison-based integer sorting algorithm where keys are first
split according to their base, or radix, into digits. Digits are then sorted starting with the
least significant digit (LSD) or most significant digit (MSD). Radix sort works by passing
through the data multiple times, each time sorting it according to the current digit. The
size of the radix (from here on referred to as k) can be arbitrarily chosen but the choice may
greatly affect the performance. A large radix results in fewer passes over the input data but
suffers from bad memory locality, while a small radix results in the opposite. The optimal
radix for a certain problem may be architecturally dependent and tests are therefore often
necessary in order to find an optimal radix.

Quick sort [7] and merge sort [8] are comparison-based sorting algorithms. They are both
so-called divide and conquer algorithms and have an average complexity in \(O(n \log n)\).

Counting sort [6, p. 194] is a noncomparison-based integer sorting algorithm that sorts
keys by counting the number of occurrences of each key. Counting sort can be realized as
follows: a histogram of all keys are first created to determine the size of each output bucket, i.e., each digit’s part of the output array (later referred to as phase 1), a prefix sum of the bucket sizes determine the starting offset of each bucket in the output array (later referred to as phase 2), and finally each key is moved to its output position (later referred to as phase 3).

Satish, Kim, Chhugani, et al. [9] argue that the right choice between comparison-based and noncomparison-based sorting algorithms for a certain problem becomes a trade-off between computational complexity and architectural efficiency. Noncomparison-based sorting algorithms have better time complexity but are often not as architecturally friendly as comparison-based sorting algorithms.

Davis [10] presents an implementation of MSD radix sort and compares this implementation with \texttt{qsort} (a function in the C standard library) and a custom implementation of quick sort. Empirical results suggest that radix sort is faster than both \texttt{qsort} and quick sort at sorting 65,536 keys of different fixed lengths and distributions.

Larriba-Pey, Jimenez, and Navarro [11] presents an approach to counting sort that, when used in combination with LSD radix sort, reduces the passes over the input vector of radix sort from $2m$ to $m + 1$ where $m$ is the number of digits. For example, radix sorting a 32-bit key with radix 256 (8 bits) gives 4 digits. By creating $4 \cdot 256$ bins, the initial counting pass can be done once for all digits instead of once per digit.

Rahman and Raman [12] present optimized versions of LSD radix sort by reducing misses in the translation-lookaside buffer (TLB) using different techniques. The TLB is a cache used to improve the translation of virtual addresses. The approach that is shown to be fastest according to empirical results is one that reduces misses in the TLB by pre-sorting keys within relatively small segments before each pass of LSD radix sort. For each pass, the input is first divided into segments that are locally sorted before the whole input is sorted. This pre-sorting approach increases the locality of subsequent radix sorts and is shown to be 1.85x faster than an optimized version of quick sort and 2.45x faster than an optimized version of merge sort when sorting 32 million uniformly distributed 32-bit integers.

Jimenez-Gonzalez, Navarro, and Larriba-Pey [13] presents a cache-conscious radix sort, which is claimed to be twice as fast as quick sort and 1.4x as fast as the \textit{explicit block transfer radix sort} presented in [12] which was claimed to be the fastest known at the time. Rahman and Raman [12] do, however, not conclude this to be the case, rather, the pre-sorting technique is shown to be superior. The presented cache conscious radix sort improves data locality of radix sort by partitioning data into subsets that fits inside certain cache levels and the partitioning is done using MSD radix sort followed by LSD radix sort.

Satish, Kim, Chhugani, et al. [9] present an analysis of comparison and noncomparison-based sorting algorithms on CPUs and GPUs. Merge sort is argued to become superior to LSD radix sort on future architectures, even though radix sort has better time complexity, due to more efficient utilization of data-level parallelism and low memory bandwidth utilization. Their analysis considers both thread-level parallelism and data-level parallelism. Their evaluation shows that for sorting $2^{16}$ to $2^{27}$ 32-bit integer keys on current CPU architecture, LSD radix sort is superior to merge sort for inputs equal to or larger than $2^{17}$. At input size $2^{27}$, the throughput of radix sort is 1.7x larger than that of merge sort. They argue that the computational complexity of merge sort will, for future architectures, not overwhelm its architectural advantages as the input size grows. Later work [14] also considers the Intel MIC architecture. Two different radix sorts are proposed. A variant called \textit{split radix sort} is shown to not perform well due to low SIMD width (the size of a SIMD register). The
idea behind the other proposed radix sort, called buffer-based radix sort, is to locally collect elements belonging to the same radix and write them to global memory once enough elements have accumulated. This makes writes to global memory larger in granularity. Straightforward implementations on cache-based architectures may perform this kind of buffering implicitly, since writes are cached and not continuously written.

Wassenberg and Sanders [15], [16] demonstrate a variant of LSD radix sort that is built upon a microarchitecture-aware variant of counting sort. They focus on integer key/value pairs of 32-bits each. A throughput within 12% of theoretical optimum for given hardware is achieved. Claims to outperform Intel’s buffer-based radix sort by a factor of 1.5 by pre-allocating buckets, using software write-combining, and something called reverse sorting. Reverse sorting means, similar to what was done in [13], to initially sort the data with one or more passes of MSD radix sort before sorting the remaining data using LSD radix sort. Pre-allocation of buckets is done in a similar fashion as in [11] and it is argued that the extra space only occupies virtual memory, i.e., address space, until it is accessed. Software write-combining allows data to be written together in a burst instead of as single bits or small chunks. Data is first placed into temporary buffers before it is written to its actual destination. This technique can be compared to the buffer-based radix sort in [9].

Curi-Quintal, Cadenas, and Megson [17] presents a fast integer sorting algorithm capable of sorting distinct integers called bit-index sort. Exhibits faster execution times than both counting sort and quick sort for input sizes between 100 and 182,000 and values between 0 and 307,199. The performance advantage relative to quick sort and counting sort increases respectively decreases as input size grows. For the largest inputs, the sequential bit-index sort is 12x faster than qsort and more than twice as fast as counting sort. Bit-index sort achieves this by using an array where each element is encoded as a single bit, assuming distinct values, and exploiting hardware instructions such as count trailing zeros (ctz) and count leading zeros (clz). A parallel version is also presented which utilizes two threads, each processing the array of encoded elements from different ends. The speedup is between 130% and 166%. Arbitrary data is not shown to be able to accompany the distinct values being sorted.

In summary, previous research indicates that a noncomparison-based sorting algorithm is superior for the integer sorting problem at hand. More specifically, this survey suggests that LSD radix sort together with counting sort is likely to perform well. A number of implementation techniques to increase the performance of radix sort have been presented, many of them having common traits. In particular, the following ideas are recurring in the literature: (1) single-pass counting sort by pre-allocating buckets, (2) pre-sorting the input using various techniques, (3) buffering writes, (4) parallelization schemes, and (5) finding the optimal radix by empirical testing.
2 Synthesis

In the literature survey in Section 1.4, a sorting algorithm that is likely to perform well for the specific sorting problem at hand was identified, together with a number of common implementation techniques. In this chapter, a library for manipulating sparse multidimensional arrays and a tailored version of radix sort, called smda_sort, is presented. The library is implemented in the C programming language and is called smda for sparse multidimensional arrays.

2.1 Extracting keys

Since the keys to be sorted are not given in a standard format (as described in Section 1.1), prior to being sorted, they have to be extracted from the packed multi-indices. Moreover, smda_sort must be able to sort according to multiple dimensions in arbitrary order. Bits representing relevant dimensions of each multi-index must therefore be extracted and shuffled to the specified order. Consider sorting according to three dimensions in the order 0, 2, 1.

Using LSD radix sort, one must start by sorting the least significant digit of the least significant dimension (dimension 1 in this case), proceeding to more significant digits and dimensions until all digits of all dimensions have been processed. By using a bit mask, all bits of a specific dimension can be extracted from a bit field using the bitwise AND operation followed by a right shift. One of the following techniques can be used if the number of bits in the chosen radix does not evenly divide the number of bits used by each dimension:

Naive approach
Extract only as many bits as possible from the current dimension.

Overlapping approach
Extract as many bits as possible from the current dimension and proceed to extract bits from subsequent dimensions according to the sort order if needed.

Rearranging approach
Rearrange bit fields prior to the first pass according to the sort order, to obtain keys in a standard format.

Note that if \( k = \log_2(\text{radix}) \), the first approach does not guarantee it will process \( k \) bits each pass, rather, it will process up to \( k \) bits each pass. The overlapping approach will process \( k \) bits per pass except possibly the last pass. The rearranging approach will either include a side effect that rearranges the bit fields of \( \omega \) according to the dimensions being sorted, require extra memory for storing the rearranged version of \( \omega \), or require an additional pass at the end to undo the initial rearrangement of the bit fields in \( \omega \). Since it is not immediately obvious which of these approaches is most efficient, all of them were implemented and evaluated. The results can be seen in Chapter 3.
2.2 Single-pass counting sort

The approach of Larriba-Pey, Jimenez, and Navarro [11] to reduce the number of passes over the input array when using radix sort in combination with counting sort is better suited to use with the sophisticated approaches of key extraction compared to the naive approach. This is because the minimum number of bits processed each pass for the naive approach is 1. This forces single-pass counting sort to allocate 32x the amount of memory compared to regular counting sort for 32-bit keys. The sophisticated approaches, in comparison, only require $32/k$ times more memory. The single-pass counting sort was implemented and evaluated for one of these sophisticated approaches, the evaluation can be seen in Section 3.2. The single-pass counting sort is, however, only applicable to sequential versions of radix sort.

2.3 Parallelization

In the literature survey, different techniques for optimizing the cache performance of radix sort using pre-sorting in various ways are presented. By parallelizing the tailored implementation of radix sort using a variation of the basic parallel algorithm [9], both speedup from using multiple threads and increased cache performance can be achieved since partitioning the input among threads has similar effects as the pre-sorting technique described by Rahman and Raman [12]. Experiments in Chapter 3 shows that the time spent in phase 2 (prefix sum) of the sequential counting sort implementation is negligible compared to its other phases, only phases 1 and 3 are therefore parallelized. Parallelization is done using OpenMP [18] and an algorithm based on [9]:

Phase 1

Each thread computes a local histogram of its partition of the input.

Phase 2

The starting offsets of all thread’s digits are calculated by the master thread using the local histograms from phase 1.

Phase 3

Each thread writes its portion of the input to the output array using the offsets calculated in phase 2.

Phase 2 of the parallel algorithm can be done in two different ways: the global offsets for each thread can either be stored in the shared histogram buffer with a stride (since each radix’s offset is now split between $p$ threads) or one could put all offsets belonging to a specific thread sequentially in the output by iterating phase 2 by the mentioned stride. The latter was chosen to increase the cache performance of phase 3.

2.4 smda_sort

smda_sort uses the naive approach to key extraction. It avoids the redundant copy at the end of odd-digit radix sorts by passing two buffers to smda_sort and informing the user which buffer the sorted data resides in upon completion. It uses the empirically determined optimal radix of $2^8$ and uses a sequential version until the input size gets large enough for a parallel version to become superior.
2.5 Library design

In this section, a brief technical description of the smda library is presented.

Data structures

The multi-indices are represented in so-called augmented form (32 bits for the key and another 32 bits for the permutation information) with a 64-bit word size. The following C type declarations are used for multi-indices and augmented multi-indices, respectively:

```c
typedef uint32_t mi_t;
typedef uint64_t augmi_t;
```

The characteristics of a tensor are represented by the following type:

```c
typedef struct smda_s {
  int order;
  unsigned size[MAX_ORDER];
  int bits[MAX_ORDER];
  int offs[MAX_ORDER];
} smda_t;
```

The `order` field represents \( N \), the order of the tensor. The array `size` represents the dimensions \( I_n \) of the tensor. The array `bits` stores the number of bits required to represent each dimension. Finally, the array `offs` represents the offset of the bit field for the corresponding index.

Functions

The smda library contains the following functions:

- `smda_create` - creates a smda_t object from order and size,
- `smda_pack` - packs an unpacked array of multi-indices,
- `smda_unpack` - unpacks a packed array of multi-indices,
- `smda_sample` - generates a random multi-index set,
- `smda_sort` - sorts an array of augmented multi-indices in one or more dimensions,
- `smda_permute` - permutes an array based on a supplied permutation.

Test suite

A test suite accompanies the library capable of testing it for correctness and measuring its performance. Performance is measured using wall clock times and correctness is ensured by comparing the result of each sorting operation with that of `qsort` from the C standard library. In contrast to `smda_sort`, however, permutation information is not generated when using `qsort` to sort the indices.

Wall clock times are measured using `gettimeofday()` from the GNU C library and the overhead of allocating memory, constructing and deconstructing threads (if needed) and so forth is all included in the execution times to reflect real world use cases. A compilation switch also exists to measure phase times within the sorting function as opposed to only the total execution time.
3 Evaluation

In this chapter, evaluations of the implementations discussed in Chapter 2 are presented.

3.1 Setup

All tests were done on a system with an Intel Core i7-4960HQ with 8 logical cores and 16 GB of 1600 MHz DDR3 memory running OS X 10.9. The test suite presented in Section 2.5 was used for all tests. Compilation was done using gcc-4.9.0 on OS X 10.9 with optimization -03 and -funroll-all-loops.

Each result presented in this chapter is based on the average measured execution times over 100 executions. For each run, random uniform data was generated within the range of possible values for the tensor being sorted and all available dimensions were sorted in random order. The order and size of the tensor being tested is presented for each test along with information such as the radix used. For each experiment (if not stated otherwise) tests are done for input sizes, i.e., non-zero elements (nnz), between 5,000 and 5,000,000. This range was arbitrarily chosen to include both very sparse and less sparse tensors.

3.2 Comparison of key extraction techniques

The first goal is to determine which approach to extracting keys and digits from the bit fields is most efficient. In order to determine this, an experiment was first done comparing the performance of different radices. Figure 3.1 shows the result of sorting a tensor of order 1 and size $2^{24}$ using the naive approach with different radices. It should be noted that this test was, knowingly, done using a large set of duplicate multi-indices in the omega array. It can be seen that a radix of 8 bits, i.e., 256, is superior compared to both smaller and larger radices. Moreover, using the customized naive radix sort appears to be superior to using qsort even for the simplest case of tensors of order 1.
The next experiment investigates whether sophisticated ways of extracting keys from the multi-indices are justified or not. In Figure 3.2, the naive approach (naive8), the overlapping approach (overl8), the overlapping approach with single-pass counting sort (overlspcs8), and the rearranging approach (rearr8) sorts a tensor with 8 dimensions of 4 bits each using a radix of 8 bits. The test is designed to favour the use of sophisticated digit extraction techniques, since the naive approach will only process 4 bits per pass, which in Figure 3.1 was shown to be more than 20% slower than processing 8 bits. According to the results in Figure 3.2, the overhead of extracting bits in sophisticated ways is substantial and the naive approach is superior to the sophisticated approaches. The rearranging approach was shown to be slowest, although its performance could likely be improved by using a different technique as described in Section 2.1. The single-pass counting sort was only applied to the overlapping approach due to the poor performance of the rearranging approach. It was, surprisingly, shown to make the overlapping approach slightly slower than without it.
To investigate the unexpected result of the experiment presented in Figure 3.2, the time spent in different phases of the counting sort algorithm was analyzed. Figure 3.3 shows the time spent in different phases for all different key extraction techniques. The overhead is also presented, this is all the time spent in the sorting routine that is not part of the counting sort algorithm. Phase 2 (prefix sum) is shown to be negligible for all techniques. This is reasonable because phase 2 iterates the number of possible key values (256 in this case) whereas phase 1 and 3 iterates the whole input array (≥ 4,000,000 in this case).

The overhead of the rearranging approach is shown to be the culprit of its bad performance although other phases are shown to be faster compared to the other techniques. The rearranging approach can therefore become competitive to the naive approach if the overhead of rearranging the omega array is reduced, something that should be further investigated in future work. Phase 1 (histogram) is not shown to be smaller for the overlapping approach with single-pass counting sort than without it, even though it makes fewer passes over the input array, the reason for this was not found.

### 3.3 The relative cost of a superfluous copy

For each pass of radix sort, all 64-bit augmented indices have to be written from one buffer to another due to the out-of-place nature of the radix sort used. Assuming the sorted result should be returned in the same buffer as it was passed within, the data has to be copied one time superfluously if the number of radix sort passes is odd. An experiment was made to measure the relative impact of this redundant copy, as shown in Figure 3.4. A tensor of order 3 and size $2^8 \times 2^8 \times 2^8$ was sorted using radix $2^8$. The relative performance cost of copying the data was shown to be considerable and increasing as the input size increases. At $nnz = 10^5$, the cost was roughly 4% of a full sort.
3.4 Optimal radix

In order to find the optimal radix value for the parallel version, an experiment was done to sort a tensor of order 1 and the same size as the radix used. The execution time was then compared to that of \texttt{qsort}. Possible bias from the phenomenon discussed in Section 3.2 is avoided by adjusting the dimension size to the radix value, while possible bias from sorting dimensions of unequal length is avoided by comparing the execution time to that of \texttt{qsort}. Radices ranging from $2^1$ to $2^{32}$ were tested, the best of which are shown in Figure 3.5. Radices $2^8$ and $2^9$ are shown to exhibit best performance, radix $2^8$ is therefore chosen since it requires a lesser amount of memory. It should be noted that this test was, knowingly, done using a large set of duplicate multi-indices in the \texttt{omega} array.

Figure 3.5: Optimal radix comparison for parallel radix sort. Performance is relative to \texttt{qsort}. A one-dimensional tensor of $\texttt{size} = \texttt{radix}$ was sorted.
3.5 Optimal number of threads

An experiment was done to find the optimal number of threads for different sizes of input. A tensor of order 3 and dimensions of size $2^8$ was sorted at inputs between 5,000 and 5,000,000. Figure 3.6 shows the speedup of the custom parallel radix sort using different number of threads. It can be seen that the optimal number of threads is dependent upon the input size. At sizes below 10$^5$ the sequential version is faster than the parallel version and should therefore be used, at sizes slightly larger than 10$^5$ the parallel version with 8 threads is faster and should be opted to be used instead.

![Figure 3.6: Speedup of the parallel custom radix sort.](image)

3.6 smda_sort benchmark

The performance of smda_sort can be seen in Figures 3.7 and 3.8. For sorting a tensor of order 3 and dimensions of size $2^8$, the sequential version is shown to be consistently around 20 times faster than qsort, as seen in Figure 3.7. The parallel version is shown to be 20 to 70 times faster, as seen in Figure 3.7. The execution time of the parallel version for sorting a larger tensor of order 4 with dimensions of size $2^8$ and with up to 50,000,000 non-zero elements can be seen in Figure 3.8. It can be seen that the execution time is linearly increased as the input size grows—a feature of noncomparison-based sorting algorithms when implemented correctly. Table 3.1 presents some of the execution times of the parallel version from Figures 3.7 and 3.8 at different values of nnz.
order = 3, dim\_size = 2^8, radix = 2^8

Figure 3.7: The performance of the parallel and sequential version of smda\_sort shown as a ratio to qsort.

order = 4, dim\_size = 2^8, radix = 2^8

Figure 3.8: The execution time (s) of the parallel version of smda\_sort.

<table>
<thead>
<tr>
<th>Figure 3.7</th>
<th>Figure 3.8</th>
<th>nnz</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.000079s</td>
<td>0.000108s</td>
<td>10,368</td>
</tr>
<tr>
<td>0.001255s</td>
<td>0.001527s</td>
<td>110,906</td>
</tr>
<tr>
<td>0.005129s</td>
<td>0.006686s</td>
<td>1,186,597</td>
</tr>
<tr>
<td>0.015864s</td>
<td>0.059545s</td>
<td>10,579,803</td>
</tr>
<tr>
<td>–</td>
<td>0.371141s</td>
<td>45,491,209</td>
</tr>
</tbody>
</table>

Table 3.1: Execution times (s) of the parallel version from Figures 3.7 and 3.8 at different values of nnz.
4 Conclusion

This thesis aims to identify fast sorting algorithms and customize them for a particular integer sorting problem related to the manipulation of sparse multidimensional arrays, often called tensors. The long-term goal is to create a high-performance library for their manipulation. A key component in such a library is to sort tensors in one or more dimensions. The purpose of this thesis is to develop the basis of such a library with emphasis on sorting.

A literature survey identified LSD radix sort together with counting sort to be the best candidate for the problem at hand along with a number of different implementation techniques. An evaluation of different implementations show that a naive approach to extracting keys from the non-standard key-format is superior to more sophisticated techniques. It also shows that one should avoid the redundant copy at the end of odd-digit, out-of-place, radix sort. A parallel version of the customized algorithm was shown to be superior to the sequential version for input sizes larger than 100,000 using 8 threads and the optimal radix was found to be $2^8$ for the hardware used.

The performance of smda_sort was tested using a tensor of order 3, dimension size of $2^8$, and with non-zero elements between 5,000 and 5,000,000. The sequential version was shown to be consistently around 20 times faster than qsort. The parallel version was shown to be 20 to 70 times faster than qsort.

The foundation of a library for manipulating sparse multidimensional arrays was developed and is called smda. The library is open source, visit timoteus.se/research for more information.

4.1 Future work

Future work should focus on extending the smda library as it is still not complete. Suggestions for future extensions include functions that can partition the multi-index set along one or more dimensions, remove duplicates of a multi-index set by adding the values of all duplicates of the same multi-index, count the number of elements with one, some or all but one index specified, and convert to and from expanded representation schemes with a fixed number of bits for all indices regardless of the size of the dimensions.

Additional optimizations to the sorting algorithms and their implementations can also be done. Multiple cache-optimizing techniques was identified in the literature. One of them was the buffer-based radix sort [9] which is thought to have potential to further increase the performance of smda_sort. The buffer-based radix sort would, however, require analysis and tests on different hardware seeing as it is architecturally dependent. In order to preserve the generality of the smda library these parameters should probably be dynamically set. Parameters such as the optimal radix and number of threads should also be tested on different hardware and, if found to notably differ, also be dynamically set.
I wish to thank my excellent supervisor Lars Karlsson for his proposal that lead to this thesis, for his unceasing support, and his meticulous feedback. Thank you!
Bibliography


